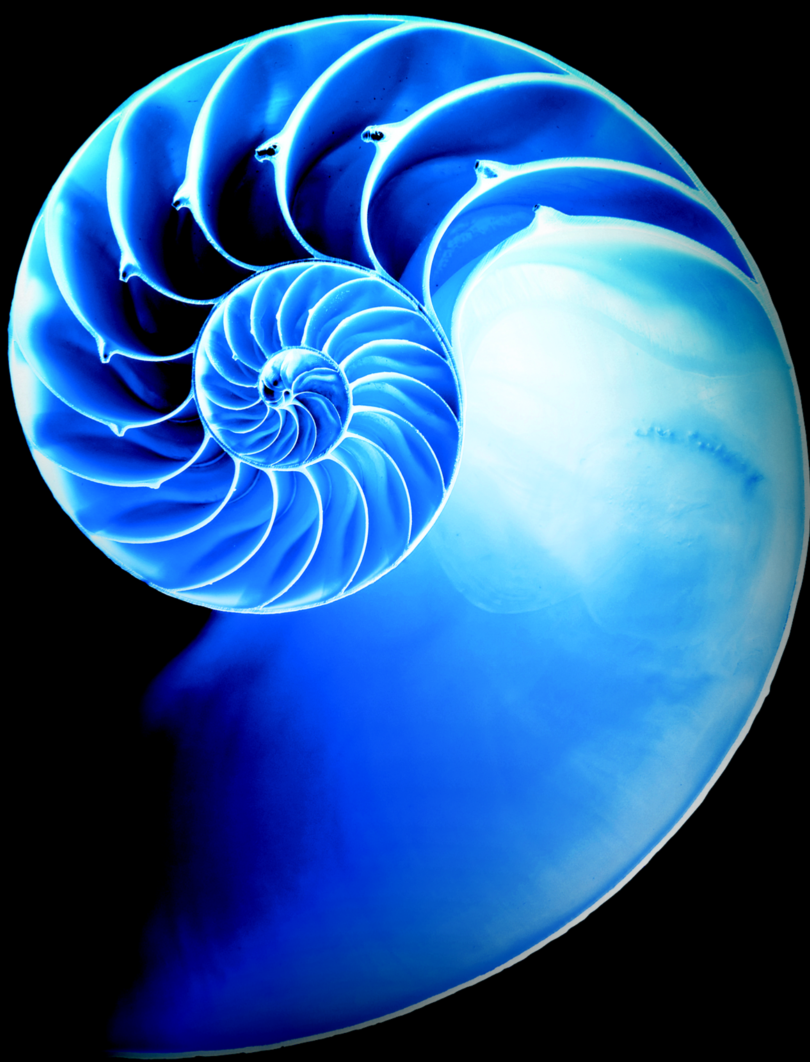
28

Networking



Objectives

In this chapter you'll:

- Implement Java networking applications by using sockets and datagrams.
- Implement Java clients and servers that communicate with one another.
- Implement network-based collaborative applications.

Outline

28.1 Introduction[1]

Java provides a number of built-in networking capabilities that make it easy to develop Internet-based and web-based applications. Java can enable programs to search the world for information and to collaborate with programs running on other computers internationally, nationally or just within an organization (subject to security constraints).

Java's fundamental networking capabilities are declared by the classes and interfaces of package **`java.net`**, through which Java offers **stream-based communications** that enable applications to view networking as streams of data. The classes and interfaces of package `java.net` also offer **packet-based communications** for transmitting individual **packets** of information—commonly used to transmit data images, audio and video over the Internet. In this chapter, we show how to communicate with packets and streams of data.

We focus on both sides of the **client/server relationship**. The **client** *requests* that some action be performed, and the **server** performs the action and *responds* to the client. A common implementation of the *request-response model* is between web browsers and web servers. When a user selects a website to browse through a browser (the client application), a request is sent to the appropriate web server (the server application). The server normally responds to the client by sending an appropriate web page to be rendered by the browser.

We introduce Java's **socket-based communications**, which enable applications to view networking as if it were *file I/O*—a program can read from a **socket** or write to a socket as simply as reading from a file or writing to a file. The socket is simply a software construct that represents one endpoint of a connection. We show how to create and manipulate *stream sockets* and *datagram sockets*. With **stream sockets**, a process establishes a **connection** to another process. While the connection is in place, data flows between the processes in continuous **streams**. Stream sockets are said to provide a **connection-oriented service**. The protocol used for transmission is the popular **TCP** (**Transmission Control Protocol**).

With **datagram sockets**, individual **packets** of information are transmitted. The protocol used—**UDP**, the **User Datagram Protocol**—is a **connectionless service** and does *not* guarantee that packets arrive in any particular *order*. With UDP, packets can even be *lost* or *duplicated*. Significant extra programming is required on your part to deal with these problems (if you choose to do so). UDP is most appropriate for network applications that do not require the error checking and reliability of TCP. Stream sockets and the TCP protocol will be more desirable for the vast majority of Java networking applications.

1. This is a legacy chapter posted as is from the book's 10th edition.

Performance Tip 28.1

Connectionless services generally offer greater performance but less reliability than connection-oriented services.

Portability Tip 28.1

TCP, UDP and related protocols enable heterogeneous computer systems (i.e., those with different processors and different operating systems) to intercommunicate.

For interested readers, we provide at

```
http://www.deitel.com/books/jhtp11
```

a case study from an older edition of this book. In the case study, we implement a client/server chat application using **multicasting**, in which a server can publish information and *many* clients can *subscribe* to it. When the server publishes information, *all* subscribers receive it.

28.2 Reading a File on a Web Server

The application in Fig. 28.1 uses Swing GUI component **`JEditorPane`** (from package `javax.swing`) to display the contents of a file on a web server. The user enters a URL in the `JTextField` at the top of the window, and the application displays the corresponding document (if it exists) in the `JEditorPane`. Class `JEditorPane` is able to render both plain text and basic HTML-formatted text, as illustrated in the two screen captures (Fig. 28.2), so this application acts as a simple web browser. The application also demonstrates how to process **`HyperlinkEvents`** when the user clicks a hyperlink in the HTML document.

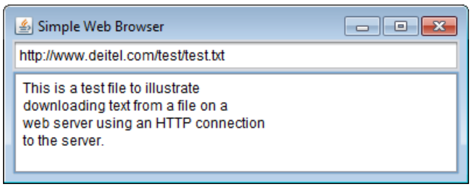
```java
// Fig. 28.1: ReadServerFile.java
// Reading a file by opening a connection through a URL.
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import javax.swing.JEditorPane;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextField;
import javax.swing.event.HyperlinkEvent;
import javax.swing.event.HyperlinkListener;

public class ReadServerFile extends JFrame
{
   private JTextField enterField; // JTextField to enter site name
   private JEditorPane contentsArea; // to display website

   // set up GUI
   public ReadServerFile()
   {
```

Fig. 28.1 | Reading a file by opening a connection through a URL. (Part 1 of 2.)

```java
      super("Simple Web Browser");

      // create enterField and register its listener
      enterField = new JTextField("Enter file URL here");
      enterField.addActionListener(
         new ActionListener()
         {
            // get document specified by user
            public void actionPerformed(ActionEvent event)
            {
               getThePage(event.getActionCommand());
            }
         }
      );

      add(enterField, BorderLayout.NORTH);

      contentsArea = new JEditorPane(); // create contentsArea
      contentsArea.setEditable(false);
      contentsArea.addHyperlinkListener(
         new HyperlinkListener()
         {
            // if user clicked hyperlink, go to specified page
            public void hyperlinkUpdate(HyperlinkEvent event)
            {
               if (event.getEventType() ==
                    HyperlinkEvent.EventType.ACTIVATED)
                  getThePage(event.getURL().toString());
            }
         }
      );

      add(new JScrollPane(contentsArea), BorderLayout.CENTER);
      setSize(400, 300); // set size of window
      setVisible(true); // show window
   }

   // load document
   private void getThePage(String location)
   {
      try // load document and display location
      {
         contentsArea.setPage(location); // set the page
         enterField.setText(location); // set the text
      }
      catch (IOException ioException)
      {
         JOptionPane.showMessageDialog(this,
            "Error retrieving specified URL", "Bad URL",
            JOptionPane.ERROR_MESSAGE);
      }
   }
}
```

Fig. 28.1 | Reading a file by opening a connection through a URL. (Part 2 of 2.)

```java
// Fig. 28.2: ReadServerFileTest.java
// Create and start a ReadServerFile.
import javax.swing.JFrame;

public class ReadServerFileTest
{
   public static void main(String[] args)
   {
      ReadServerFile application = new ReadServerFile();
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```



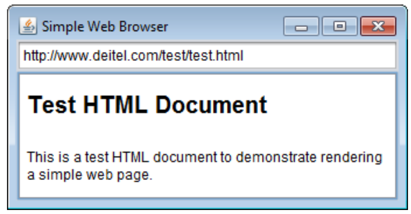


Fig. 28.2 | Test class for `ReadServerFile`.

The application class `ReadServerFile` contains `JTextField enterField`, in which the user enters the URL of the file to read and `JEditorPane contentsArea` to display the file's contents. When the user presses the *Enter* key in `enterField`, the application calls method `actionPerformed` (lines 31–34). Line 33 uses `ActionEvent` method `getActionCommand` to get the `String` the user input in the `JTextField` and passes the `String` to utility method `getThePage` (lines 61–74).

Line 65 invokes `JEditorPane` method **`setPage`** to download the document specified by `location` and display it in the `JEditorPane`. If there's an error downloading the document, method `setPage` throws an `IOException`. Also, if an invalid URL is specified, a `MalformedURLException` (a subclass of `IOException`) occurs. If the document loads successfully, line 66 displays the current location in `enterField`.

Typically, an HTML document contains **hyperlinks** that, when clicked, provide quick access to another document on the web. If a `JEditorPane` contains an HTML document and the user clicks a hyperlink, the `JEditorPane` generates a **`HyperlinkEvent`** (package `javax.swing.event`) and notifies all registered **`HyperlinkListeners`** (package `javax.swing.event`) of that event. Lines 42–53 register a `HyperlinkListener` to handle `HyperlinkEvents`. When a `HyperlinkEvent` occurs, the program calls method **`hyperlinkUpdate`** (lines 46–51). Lines 48–49 use `HyperlinkEvent` method **`getEventType`** to determine the type of the `HyperlinkEvent`. Class `HyperlinkEvent` contains a `public` nested class called **`EventType`** that declares three `static EventType` objects, which represent the hyperlink event types. **`ACTIVATED`** indicates that the user clicked a hyperlink to change web pages, **`ENTERED`** indicates that the user moved the mouse over a hyperlink and **`EXITED`** indicates that the user moved the mouse away from a hyperlink. If a hyperlink was `ACTIVATED`, line 50 uses `HyperlinkEvent` method **`getURL`** to obtain the `URL` represented by the hyperlink. Method `toString` converts the returned `URL` to a `String` that can be passed to utility method `getThePage`.

Look-and-Feel Observation 28.1

A `JEditorPane` generates `HyperlinkEvent`s only if it's uneditable.

28.3 Establishing a Simple Server Using Stream Sockets

The two examples discussed so far use *high-level* Java networking capabilities to communicate between applications. In the examples, it was not your responsibility to establish the connection between a client and a server. The first program relied on the web browser to communicate with a web server. The second program relied on a `JEditorPane` to perform the connection. This section begins our discussion of creating your own applications that can communicate with one another.

Step 1: Create a `ServerSocket`

Establishing a simple server in Java requires five steps. *Step 1* is to create a **`ServerSocket`** object. A call to the `ServerSocket` constructor, such as

```
ServerSocket server = new ServerSocket(portNumber, queueLength);
```

registers an available TCP **port number** and specifies the maximum number of clients that can wait to connect to the server (i.e., the **queue length**). The port number is used by clients to locate the server application on the server computer. This is often called the **handshake point**. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients—a process known as **binding the server to the port**. Each client will ask to connect to the server on this **port**. Only one application at a time can be bound to a specific port on the server.

Software Engineering Observation 28.1

Port numbers can be between 0 and 65,535. Most operating systems reserve port numbers below 1024 for system services (e.g., e-mail and World Wide Web servers). Generally, these ports should not be specified as connection ports in user programs. In fact, some operating systems require special access privileges to bind to port numbers below 1024.

Step 2: Wait for a Connection

Programs manage each client connection with a **`Socket`** object. In *Step 2*, the server listens indefinitely (or **blocks**) for an attempt by a client to connect. To listen for a client connection, the program calls `ServerSocket` method **`accept`**, as in

```
Socket connection = server.accept();
```

which returns a `Socket` when a connection with a client is established. The `Socket` allows the server to interact with the client. The interactions with the client actually occur at a different server port from the *handshake point*. This allows the port specified in *Step 1* to be used again in a multithreaded server to accept another client connection. We demonstrate this concept in Section 28.7.

Step 3: Get the `Socket`*'s I/O Streams*

Step 3 is to get the `OutputStream` and `InputStream` objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to the client via an `OutputStream` and receives information from the client via an `InputStream`. The server invokes method **`getOutputStream`** on the `Socket` to get a reference to the `Socket`'s `OutputStream` and invokes method **`getInputStream`** on the `Socket` to get a reference to the `Socket`'s `InputStream`.

The stream objects can be used to send or receive individual bytes or sequences of bytes with the `OutputStream`'s method `write` and the `InputStream`'s method `read`, respectively. Often it's useful to send or receive values of primitive types (e.g., `int` and `double`) or `Serializable` objects (e.g., `String`s or other serializable types) rather than sending bytes. In this case, we can use the techniques discussed in Chapter 15 to wrap other stream types (e.g., `ObjectOutputStream` and `ObjectInputStream`) around the `OutputStream` and `InputStream` associated with the `Socket`. For example,

```
ObjectInputStream input =
   new ObjectInputStream(connection.getInputStream());
ObjectOutputStream output =
   new ObjectOutputStream(connection.getOutputStream());
```

The beauty of establishing these relationships is that whatever the server writes to the `ObjectOutputStream` is sent via the `OutputStream` and is available at the client's `InputStream`, and whatever the client writes to its `OutputStream` (with a corresponding `ObjectOutputStream`) is available via the server's `InputStream`. The transmission of the data over the network is seamless and is handled completely by Java.

Step 4: Perform the Processing

Step 4 is the *processing* phase, in which the server and the client communicate via the `OutputStream` and `InputStream` objects.

Step 5: Close the Connection

In *Step 5*, when the transmission is complete, the server closes the connection by invoking the **`close`** method on the streams and on the `Socket`.

Software Engineering Observation 28.2

With sockets, network I/O appears to Java programs to be similar to sequential file I/O. Sockets hide much of the complexity of network programming.

Software Engineering Observation 28.3

A multithreaded server can take the `Socket` *returned by each call to* `accept` *and create a new thread that manages network I/O across that* `Socket`*. Alternatively, a multithreaded server can maintain a pool of threads (a set of already existing threads) ready to manage network I/O across the new* `Socket`*s as they're created. These techniques enable multithreaded servers to manage many simultaneous client connections.*

Performance Tip 28.2

In high-performance systems in which memory is abundant, a multithreaded server can create a pool of threads that can be assigned quickly to handle network I/O for new `Sockets` *as they're created. Thus, when the server receives a connection, it* need not incur thread-creation overhead. *When the connection is closed, the thread is returned to the pool for reuse.*

28.4 Establishing a Simple Client Using Stream Sockets

Establishing a simple client in Java requires four steps.

Step 1: Create a **Socket** *to Connect to the sServer*

In *Step 1*, we create a Socket to connect to the server. The Socket constructor establishes the connection. For example, the statement

```
Socket connection = new Socket(serverAddress, port);
```

uses the Socket constructor with two arguments—the server's address (*serverAddress*) and the *port* number. If the connection attempt is successful, this statement returns a Socket. A connection attempt that fails throws an instance of a subclass of IOException, so many programs simply catch IOException. An **UnknownHostException** occurs specifically when the system is unable to resolve the server name specified in the call to the Socket constructor to a corresponding IP address.

Step 2: Get the **Socket***'s I/O Streams*

In *Step 2*, the client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream. As we mentioned in the preceding section, we can use the techniques of Chapter 15 to wrap other stream types around the InputStream and OutputStream associated with the Socket. If the server is sending information in the form of actual types, the client should receive the information in the same format. Thus, if the server sends values with an ObjectOutputStream, the client should read those values with an ObjectInputStream.

Step 3: Perform the Processing

Step 3 is the processing phase in which the client and the server communicate via the InputStream and OutputStream objects.

Step 4: Close the Connection

In *Step 4*, the client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket. The client must determine when the server is finished sending information so that it can call close to close the Socket connection. For example, the InputStream method read returns the value –1 when it detects end-of-stream (also called EOF—end-of-file). If an ObjectInputStream reads information from the server, an EOFException occurs when the client attempts to read a value from a stream on which end-of-stream is detected.

28.5 Client/Server Interaction with Stream Socket Connections

Figures 28.3 and 28.5 use stream sockets, ObjectInputStream and ObjectOutputStream to demonstrate a simple **client/server chat application**. The server waits for a client connection attempt. When a client connects to the server, the server application sends the client a String object (recall that Strings are Serializable objects) indicating that the connection was successful. Then the client displays the message. The client and server applications each provide text fields that allow the user to type a message and send it to the other application. When the client or the server sends the String "TERMINATE", the connection terminates. Then the server waits for the next client to connect. The declaration of class `Server` appears in Fig. 28.3. The declaration of class `Client` appears in Fig. 28.5. The screen captures showing the execution between the client and the server are shown in Fig. 28.6.

Server *Class*

`Server`'s constructor (Fig. 28.3, lines 30–55) creates the server's GUI, which contains a `JTextField` and a `JTextArea`. `Server` displays its output in the `JTextArea`. When the `main` method (lines 6–11 of Fig. 28.4) executes, it creates a `Server` object, specifies the window's default close operation and calls method `runServer` (Fig. 28.3, lines 57–86).

```
1  // Fig. 28.3: Server.java
2  // Server portion of a client/server stream-socket connection.
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.net.ServerSocket;
8  import java.net.Socket;
9  import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Server extends JFrame
19 {
20    private JTextField enterField; // inputs message from user
21    private JTextArea displayArea; // display information to user
22    private ObjectOutputStream output; // output stream to client
23    private ObjectInputStream input; // input stream from client
24    private ServerSocket server; // server socket
25    private Socket connection; // connection to client
26    private int counter = 1; // counter of number of connections
27
28    // set up GUI
29    public Server()
30    {
31       super("Server");
32
33       enterField = new JTextField(); // create enterField
34       enterField.setEditable(false);
35       enterField.addActionListener(
36          new ActionListener()
37          {
38             // send message to client
39             public void actionPerformed(ActionEvent event)
40             {
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 1 of 4.)

```
               sendData(event.getActionCommand());
               enterField.setText("");
            }
         }
      );

      add(enterField, BorderLayout.NORTH);

      displayArea = new JTextArea(); // create displayArea
      add(new JScrollPane(displayArea), BorderLayout.CENTER);

      setSize(300, 150); // set size of window
      setVisible(true); // show window
   }

   // set up and run server
   public void runServer()
   {
      try // set up server to receive connections; process connections
      {
         server = new ServerSocket(12345, 100); // create ServerSocket

         while (true)
         {
            try
            {
               waitForConnection(); // wait for a connection
               getStreams(); // get input & output streams
               processConnection(); // process connection
            }
            catch (EOFException eofException)
            {
               displayMessage("\nServer terminated connection");
            }
            finally
            {
               closeConnection(); // close connection
               ++counter;
            }
         }
      }
      catch (IOException ioException)
      {
         ioException.printStackTrace();
      }
   }

   // wait for connection to arrive, then display connection info
   private void waitForConnection() throws IOException
   {
      displayMessage("Waiting for connection\n");
      connection = server.accept(); // allow server to accept connection
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 2 of 4.)

```java
      displayMessage("Connection " + counter + " received from: " +
         connection.getInetAddress().getHostName());
   }

   // get streams to send and receive data
   private void getStreams() throws IOException
   {
      // set up output stream for objects
      output = new ObjectOutputStream(connection.getOutputStream());
      output.flush(); // flush output buffer to send header information

      // set up input stream for objects
      input = new ObjectInputStream(connection.getInputStream());

      displayMessage("\nGot I/O streams\n");
   }

   // process connection with client
   private void processConnection() throws IOException
   {
      String message = "Connection successful";
      sendData(message); // send connection successful message

      // enable enterField so server user can send messages
      setTextFieldEditable(true);

      do // process messages sent from client
      {
         try // read message and display it
         {
            message = (String) input.readObject(); // read new message
            displayMessage("\n" + message); // display message
         }
         catch (ClassNotFoundException classNotFoundException)
         {
            displayMessage("\nUnknown object type received");
         }

      } while (!message.equals("CLIENT>>> TERMINATE"));
   }

   // close streams and socket
   private void closeConnection()
   {
      displayMessage("\nTerminating connection\n");
      setTextFieldEditable(false); // disable enterField

      try
      {
         output.close(); // close output stream
         input.close(); // close input stream
         connection.close(); // close socket
      }
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 3 of 4.)

```java
      catch (IOException ioException)
      {
         ioException.printStackTrace();
      }
   }

   // send message to client
   private void sendData(String message)
   {
      try // send object to client
      {
         output.writeObject("SERVER>>> " + message);
         output.flush(); // flush output to client
         displayMessage("\nSERVER>>> " + message);
      }
      catch (IOException ioException)
      {
         displayArea.append("\nError writing object");
      }
   }

   // manipulates displayArea in the event-dispatch thread
   private void displayMessage(final String messageToDisplay)
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates displayArea
            {
               displayArea.append(messageToDisplay); // append message
            }
         }
      );
   }

   // manipulates enterField in the event-dispatch thread
   private void setTextFieldEditable(final boolean editable)
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // sets enterField's editability
            {
               enterField.setEditable(editable);
            }
         }
      );
   }
}
```

Fig. 28.3 | Server portion of a client/server stream-socket connection. (Part 4 of 4.)

```java
// Fig. 28.4: ServerTest.java
// Test the Server application.
import javax.swing.JFrame;

public class ServerTest
{
   public static void main(String[] args)
   {
      Server application = new Server(); // create server
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.runServer(); // run server application
   }
}
```

Fig. 28.4 | Test class for `Server`.

Method `runServer`

Method `runServer` (Fig. 28.3, lines 57–86) sets up the server to receive a connection and processes one connection at a time. Line 61 creates a `ServerSocket` called `server` to wait for connections. The `ServerSocket` listens for a connection from a client at port `12345`. The second argument to the constructor is the number of connections that can wait in a queue to connect to the server (`100` in this example). If the queue is full when a client attempts to connect, the server refuses the connection.

Common Programming Error 28.1

Specifying a port that's already in use or specifying an invalid port number when creating a `ServerSocket` *results in a* **`BindException`**.

Line 67 calls method `waitForConnection` (declared at lines 89–95) to wait for a client connection. After the connection is established, line 68 calls method `getStreams` (declared at lines 98–108) to obtain references to the connection's streams. Line 69 calls method `processConnection` (declared at lines 111–132) to send the initial connection message to the client and to process all messages received from the client. The `finally` block (lines 75–79) terminates the client connection by calling method `closeConnection` (lines 135–150), even if an exception occurs. These methods call `displayMessage` (lines 168–179), which uses the event-dispatch thread to display messages in the application's `JTextArea`. **`SwingUtilities`** method **`invokeLater`** receives a `Runnable` object as its argument and places it into the event-dispatch thread for execution. This ensures that we don't modify a GUI component from a thread other than the event-dispatch thread, which is important since *Swing GUI components are not thread safe*. We use a similar technique in method `setTextFieldEditable` (lines 182–193), to set the editability of `enterField`. For more information on interface `Runnable`, see Chapter 23.

Method `waitForConnection`

Method `waitForConnection` (lines 89–95) uses `ServerSocket` method `accept` (line 92) to wait for a connection from a client. When a connection occurs, the resulting `Socket` is assigned to `connection`. Method `accept` blocks until a connection is received (i.e., the thread in which `accept` is called stops executing until a client connects). Lines 93–94 output the host name of the computer that made the connection. `Socket` method **`getInet-`**

Address returns an **InetAddress** (package `java.net`) containing information about the client computer. `InetAddress` method **getHostName** returns the host name of the client computer. For example, a special IP address (**127.0.0.1**) and host name (**localhost**) are useful for testing networking applications on your local computer (this is also known as the **loopback address**). If `getHostName` is called on an `InetAddress` containing `127.0.0.1`, the corresponding host name returned by the method will be `localhost`.

Method getStreams

Method `getStreams` (lines 98–108) obtains the `Socket`'s streams and uses them to initialize an `ObjectOutputStream` (line 101) and an `ObjectInputStream` (line 105), respectively. Note the call to `ObjectOutputStream` method **flush** at line 102. This statement causes the `ObjectOutputStream` on the server to send a **stream header** to the corresponding client's `ObjectInputStream`. The stream header contains such information as the version of object serialization being used to send objects. This information is required by the `ObjectInputStream` so that it can prepare to receive those objects correctly.

Software Engineering Observation 28.4

When using `ObjectOutputStream` and `ObjectInputStream` to send and receive data over a network connection, always create the `ObjectOutputStream` first and `flush` the stream so that the client's `ObjectInputStream` can prepare to receive the data. This is required for networking applications that communicate using `ObjectOutputStream` and `ObjectInputStream`.

Performance Tip 28.3

A computer's I/O components are typically much slower than its memory. Output buffers are used to increase the efficiency of an application by sending larger amounts of data fewer times, reducing the number of times an application accesses the computer's I/O components.

Method processConnection

Line 114 of method `processConnection` (lines 111–132) calls method `sendData` to send `"SERVER>>> Connection successful"` as a `String` to the client. The loop at lines 119–131 executes until the server receives the message `"CLIENT>>> TERMINATE"`. Line 123 uses `ObjectInputStream` method `readObject` to read a `String` from the client. Line 124 invokes method `displayMessage` to append the message to the `JTextArea`.

Method closeConnection

When the transmission is complete, method `processConnection` returns, and the program calls method `closeConnection` (lines 135–150) to close the streams associated with the `Socket` and close the `Socket`. Then the server waits for the next connection attempt from a client by continuing with line 67 at the beginning of the `while` loop.

`Server` receives a connection, processes it, closes it and waits for the next connection. A more likely scenario would be a `Server` that receives a connection, sets it up to be processed as a separate thread of execution, then immediately waits for new connections. The separate threads that process existing connections can continue to execute while the `Server` concentrates on new connection requests. This makes the server more efficient, because multiple client requests can be processed concurrently. We demonstrate a *multithreaded server* in Section 28.7.

Processing User Interactions

When the user of the server application enters a `String` in the text field and presses the *Enter* key, the program calls method `actionPerformed` (lines 39–43), which reads the `String` from the text field and calls utility method `sendData` (lines 153–165) to send the `String` to the client. Method `sendData` writes the object, flushes the output buffer and appends the same `String` to the text area in the server window. It's not necessary to invoke `displayMessage` to modify the text area here, because method `sendData` is called from an event handler—thus, `sendData` executes as part of the *event-dispatch thread*.

`Client` *Class*

Like class `Server`, class `Client`'s constructor (Fig. 28.5, lines 29–56) creates the GUI of the application (a `JTextField` and a `JTextArea`). `Client` displays its output in the text area. When method `main` (lines 7–19 of Fig. 28.6) executes, it creates an instance of class `Client`, specifies the window's default close operation and calls method `runClient` (Fig. 28.5, lines 59–79). In this example, you can execute the client from any computer on the Internet and specify the IP address or host name of the server computer as a command-line argument to the program. For example, the command

```
java Client 192.168.1.15
```

attempts to connect to the `Server` on the computer with IP address `192.168.1.15`.

```
1  // Fig. 28.5: Client.java
2  // Client portion of a stream-socket connection between client and server.
3  import java.io.EOFException;
4  import java.io.IOException;
5  import java.io.ObjectInputStream;
6  import java.io.ObjectOutputStream;
7  import java.net.InetAddress;
8  import java.net.Socket;
9  import java.awt.BorderLayout;
10 import java.awt.event.ActionEvent;
11 import java.awt.event.ActionListener;
12 import javax.swing.JFrame;
13 import javax.swing.JScrollPane;
14 import javax.swing.JTextArea;
15 import javax.swing.JTextField;
16 import javax.swing.SwingUtilities;
17
18 public class Client extends JFrame
19 {
20    private JTextField enterField; // enters information from user
21    private JTextArea displayArea; // display information to user
22    private ObjectOutputStream output; // output stream to server
23    private ObjectInputStream input; // input stream from server
24    private String message = ""; // message from server
25    private String chatServer; // host server for this application
26    private Socket client; // socket to communicate with server
27
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 1 of 5.)

```
   // initialize chatServer and set up GUI
   public Client(String host)
   {
      super("Client");

      chatServer = host; // set server to which this client connects

      enterField = new JTextField(); // create enterField
      enterField.setEditable(false);
      enterField.addActionListener(
         new ActionListener()
         {
            // send message to server
            public void actionPerformed(ActionEvent event)
            {
               sendData(event.getActionCommand());
               enterField.setText("");
            }
         }
      );

      add(enterField, BorderLayout.NORTH);

      displayArea = new JTextArea(); // create displayArea
      add(new JScrollPane(displayArea), BorderLayout.CENTER);

      setSize(300, 150); // set size of window
      setVisible(true); // show window
   }

   // connect to server and process messages from server
   public void runClient()
   {
      try // connect to server, get streams, process connection
      {
         connectToServer(); // create a Socket to make connection
         getStreams(); // get the input and output streams
         processConnection(); // process connection
      }
      catch (EOFException eofException)
      {
         displayMessage("\nClient terminated connection");
      }
      catch (IOException ioException)
      {
         ioException.printStackTrace();
      }
      finally
      {
         closeConnection(); // close connection
      }
   }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 2 of 5.)

```

   // connect to server
   private void connectToServer() throws IOException
   {
      displayMessage("Attempting connection\n");

      // create Socket to make connection to server
      client = new Socket(InetAddress.getByName(chatServer), 12345);

      // display connection information
      displayMessage("Connected to: " +
         client.getInetAddress().getHostName());
   }

   // get streams to send and receive data
   private void getStreams() throws IOException
   {
      // set up output stream for objects
      output = new ObjectOutputStream(client.getOutputStream());
      output.flush(); // flush output buffer to send header information

      // set up input stream for objects
      input = new ObjectInputStream(client.getInputStream());

      displayMessage("\nGot I/O streams\n");
   }

   // process connection with server
   private void processConnection() throws IOException
   {
      // enable enterField so client user can send messages
      setTextFieldEditable(true);

      do // process messages sent from server
      {
         try // read message and display it
         {
            message = (String) input.readObject(); // read new message
            displayMessage("\n" + message); // display message
         }
         catch (ClassNotFoundException classNotFoundException)
         {
            displayMessage("\nUnknown object type received");
         }

      } while (!message.equals("SERVER>>> TERMINATE"));
   }

   // close streams and socket
   private void closeConnection()
   {
      displayMessage("\nClosing connection");
      setTextFieldEditable(false); // disable enterField
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 3 of 5.)

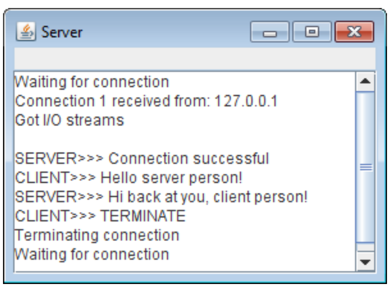
```java

      try
      {
         output.close(); // close output stream
         input.close(); // close input stream  1
         client.close(); // close socket
      }
      catch (IOException ioException)
      {
         ioException.printStackTrace();
      }
   }

   // send message to server
   private void sendData(String message)
   {
      try // send object to server
      {
         output.writeObject("CLIENT>>> " + message);
         output.flush(); // flush data to output
         displayMessage("\nCLIENT>>> " + message);
      }
      catch (IOException ioException)
      {
         displayArea.append("\nError writing object");
      }
   }

   // manipulates displayArea in the event-dispatch thread
   private void displayMessage(final String messageToDisplay)
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates displayArea
            {
               displayArea.append(messageToDisplay);
            }
         }
      );
   }

   // manipulates enterField in the event-dispatch thread
   private void setTextFieldEditable(final boolean editable)
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // sets enterField's editability
            {
               enterField.setEditable(editable);
            }
         }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 4 of 5.)

```
186          );
187       }
188    }
```

Fig. 28.5 | Client portion of a stream-socket connection between client and server. (Part 5 of 5.)

```
// Fig. 28.6: ClientTest.java
// Class that tests the Client.
import javax.swing.JFrame;

public class ClientTest
{
   public static void main(String[] args)
   {
      Client application; // declare client application

      // if no command line args
      if (args.length == 0)
         application = new Client("127.0.0.1"); // connect to localhost
      else
         application = new Client(args[0]); // use args to connect

      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.runClient(); // run client application
   }
}
```



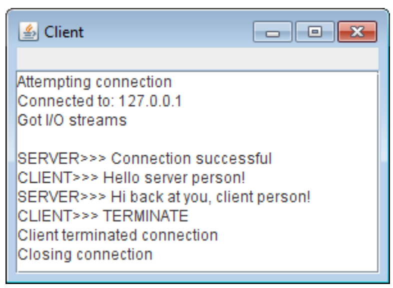


Fig. 28.6 | Class that tests the `Client`.

Method `runClient`

`Client` method `runClient` (Fig. 28.5, lines 59–79) sets up the connection to the server, processes messages received from the server and closes the connection when communication is complete. Line 63 calls method `connectToServer` (declared at lines 82–92) to perform the connection. After connecting, line 64 calls method `getStreams` (declared at lines 95–105) to obtain references to the `Socket`'s stream objects. Then line 65 calls method `processConnection` (declared at lines 108–126) to receive and display messages sent from the server. The `finally` block (lines 75–78) calls `closeConnection` (lines 129–144) to close the streams and the `Socket` even if an exception occurred. Method `displayMessage` (lines 162–173) is called from these methods to use the event-dispatch thread to display messages in the application's text area.

Method `connectToServer`

Method `connectToServer` (lines 82–92) creates a `Socket` called `client` (line 87) to establish a connection. The arguments to the `Socket` constructor are the IP address of the server computer and the port number (12345) where the server application is awaiting client connections. In the first argument, `InetAddress static` method **`getByName`** returns an `InetAddress` object containing the IP address specified as a command-line argument to the application (or `127.0.0.1` if none was specified). Method `getByName` can receive a `String` containing either the actual IP address or the host name of the server. The first argument also could have been written other ways. For the `localhost` address `127.0.0.1`, the first argument could be specified with either of the following expressions:

```
InetAddress.getByName("localhost")
InetAddress.getLocalHost()
```

Other versions of the `Socket` constructor receive the IP address or host name as a `String`. The first argument could have been specified as the IP address `"127.0.0.1"` or the host name `"localhost"`. We chose to demonstrate the client/server relationship by connecting between applications on the same computer (`localhost`). Normally, this first argument would be the IP address of another computer. The `InetAddress` object for another computer can be obtained by specifying the computer's IP address or host name as the argument to `InetAddress` method `getByName`. The `Socket` constructor's second argument is the server port number. This *must* match the port number at which the server is waiting for connections (called the *handshake point*). Once the connection is made, lines 90–91 display a message in the text area indicating the name of the server computer to which the client has connected.

The `Client` uses an `ObjectOutputStream` to send data to the server and an `ObjectInputStream` to receive data from the server. Method `getStreams` (lines 95–105) creates the `ObjectOutputStream` and `ObjectInputStream` objects that use the streams associated with the `client` socket.

Methods `processConnection` *and* `closeConnection`

Method `processConnection` (lines 108–126) contains a loop that executes until the client receives the message `"SERVER>>> TERMINATE"`. Line 117 reads a `String` object from the server. Line 118 invokes `displayMessage` to append the message to the text area. When the transmission is complete, method `closeConnection` (lines 129–144) closes the streams and the `Socket`.

Processing User Interactions

When the client application user enters a `String` in the text field and presses *Enter*, the program calls method `actionPerformed` (lines 41–45) to read the `String`, then invokes utility method `sendData` (147–159) to send the `String` to the server. Method `sendData` writes the object, flushes the output buffer and appends the same `String` to the client window's `JTextArea`. Once again, it's not necessary to invoke utility method `displayMessage` to modify the text area here, because method `sendData` is called from an event handler.

28.6 Datagrams: Connectionless Client/Server Interaction

We've been discussing connection-oriented, streams-based transmission. Now we consider **connectionless transmission with datagrams**.

Connection-oriented transmission is like the telephone system in which you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for your phone call, *even when you're not talking*.

Connectionless transmission with datagrams is more like the way mail is carried via the postal service. If a large message will not fit in one envelope, you break it into separate pieces that you place in sequentially numbered envelopes. All of the letters are then mailed at once. The letters could arrive *in order*, *out of order* or *not at all* (the last case is rare). The person at the receiving end *reassembles* the pieces into sequential order before attempting to make sense of the message.

If your message is small enough to fit in one envelope, you need not worry about the "out-of-sequence" problem, but it's still possible that your message might not arrive. One advantage of datagrams over postal mail is that duplicates of datagrams can arrive at the receiving computer.

Figures 28.7–28.10 use datagrams to send packets of information via the User Datagram Protocol (UDP) between a client application and a server application. In the `Client` application (Fig. 28.9), the user types a message into a text field and presses *Enter*. The program converts the message into a `byte` array and places it in a datagram packet that's sent to the server. The `Server` (Figs. 28.7–28.8) receives the packet and displays the information in it, then **echoes** the packet back to the client. Upon receiving the packet, the client displays the information it contains.

`Server` Class

Class `Server` (Fig. 28.7) declares two **`DatagramPacket`s** that the server uses to send and receive information and one **`DatagramSocket`** that sends and receives the packets. The constructor (lines 19–37), which is called from `main` (Fig. 28.8, lines 7–12), creates the GUI in which the packets of information will be displayed. Line 30 creates the `DatagramSocket` in a `try` block. Line 30 in Fig. 28.7 uses the `DatagramSocket` constructor that takes an integer port-number argument (`5000` in this example) to bind the server to a port where it can receive packets from clients. `Client`s sending packets to this `Server` specify the same port number in the packets they send. A **`SocketException`** is thrown if the `DatagramSocket` constructor fails to bind the `DatagramSocket` to the specified port.

Common Programming Error 28.2

Specifying a port that's already in use or specifying an invalid port number when creating a `DatagramSocket` results in a `SocketException`.

```
1  // Fig. 28.7: Server.java
2  // Server side of connectionless client/server computing with datagrams.
3  import java.io.IOException;
4  import java.net.DatagramPacket;
5  import java.net.DatagramSocket;
6  import java.net.SocketException;
7  import java.awt.BorderLayout;
8  import javax.swing.JFrame;
9  import javax.swing.JScrollPane;
10 import javax.swing.JTextArea;
```

Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 1 of 3.)

```java
import javax.swing.SwingUtilities;

public class Server extends JFrame
{
   private JTextArea displayArea; // displays packets received
   private DatagramSocket socket; // socket to connect to client

   // set up GUI and DatagramSocket
   public Server()
   {
      super("Server");

      displayArea = new JTextArea(); // create displayArea
      add(new JScrollPane(displayArea), BorderLayout.CENTER);
      setSize(400, 300); // set size of window
      setVisible(true); // show window

      try // create DatagramSocket for sending and receiving packets
      {
         socket = new DatagramSocket(5000);
      }
      catch (SocketException socketException)
      {
         socketException.printStackTrace();
         System.exit(1);
      }
   }

   // wait for packets to arrive, display data and echo packet to client
   public void waitForPackets()
   {
      while (true)
      {
         try // receive packet, display contents, return copy to client
         {
            byte[] data = new byte[100]; // set up packet
            DatagramPacket receivePacket =
               new DatagramPacket(data, data.length);

            socket.receive(receivePacket); // wait to receive packet

            // display information from received packet
            displayMessage("\nPacket received:" +
               "\nFrom host: " + receivePacket.getAddress() +
               "\nHost port: " + receivePacket.getPort() +
               "\nLength: " + receivePacket.getLength() +
               "\nContaining:\n\t" + new String(receivePacket.getData(),
                  0, receivePacket.getLength()));

            sendPacketToClient(receivePacket); // send packet to client
         }
         catch (IOException ioException)
         {
```

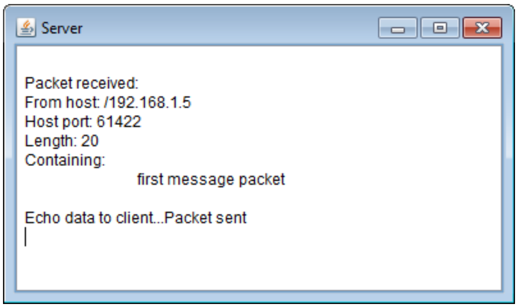
Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 2 of 3.)

```
               displayMessage(ioException + "\n");
               ioException.printStackTrace();
            }
         }
      }

      // echo packet to client
      private void sendPacketToClient(DatagramPacket receivePacket)
         throws IOException
      {
         displayMessage("\n\nEcho data to client...");

         // create packet to send
         DatagramPacket sendPacket = new DatagramPacket(
            receivePacket.getData(), receivePacket.getLength(),
            receivePacket.getAddress(), receivePacket.getPort());

         socket.send(sendPacket); // send packet to client
         displayMessage("Packet sent\n");
      }

      // manipulates displayArea in the event-dispatch thread
      private void displayMessage(final String messageToDisplay)
      {
         SwingUtilities.invokeLater(
            new Runnable()
            {
               public void run() // updates displayArea
               {
                  displayArea.append(messageToDisplay); // display message
               }
            }
         );
      }
   }
```

Fig. 28.7 | Server side of connectionless client/server computing with datagrams. (Part 3 of 3.)

```
// Fig. 28.8: ServerTest.java
// Class that tests the Server.
import javax.swing.JFrame;

public class ServerTest
{
   public static void main(String[] args)
   {
      Server application = new Server(); // create server
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.waitForPackets(); // run server application
   }
}
```

Fig. 28.8 | Class that tests the Server. (Part 1 of 2.)



`Server` window after packet of data is received from `Client`

Fig. 28.8 | Class that tests the `Server`. (Part 2 of 2.)

Method `waitForPackets`

`Server` method `waitForPackets` (Fig. 28.7, lines 40–68) uses an infinite loop to wait for packets to arrive at the `Server`. Lines 47–48 create a `DatagramPacket` in which a received packet of information can be stored. The `DatagramPacket` constructor for this purpose receives two arguments—a `byte` array in which the data will be stored and the length of the array. Line 50 uses `DatagramSocket` method **`receive`** to wait for a packet to arrive at the `Server`. Method `receive` blocks until a packet arrives, then stores the packet in its `DatagramPacket` argument. The method throws an `IOException` if an error occurs while receiving a packet.

Method `displayMessage`

When a packet arrives, lines 53–58 call method `displayMessage` (declared at lines 86–97) to append the packet's contents to the text area. `DatagramPacket` method **`getAddress`** (line 54) returns an `InetAddress` object containing the IP address of the computer from which the packet was sent. Method **`getPort`** (line 55) returns an integer specifying the port number through which the client computer sent the packet. Method **`getLength`** (line 56) returns an integer representing the number of bytes of data received. Method **`getData`** (line 57) returns a `byte` array containing the data. Lines 57–58 initialize a `String` object using a three-argument constructor that takes a `byte` array, the offset and the length. This `String` is then appended to the text to display.

Method `sendPacketToClient`

After displaying a packet, line 60 calls method `sendPacketToClient` (declared at lines 71–83) to create a new packet and send it to the client. Lines 77–79 create a `DatagramPacket` and pass four arguments to its constructor. The first argument specifies the `byte` array to send. The second argument specifies the number of bytes to send. The third argument specifies the client computer's IP address, to which the packet will be sent. The fourth argument specifies the port where the client is waiting to receive packets. Line 81 sends the packet over the network. Method **`send`** of `DatagramSocket` throws an `IOException` if an error occurs while sending a packet.

`Client` *Class*

The `Client` (Figs. 28.9–28.10) works similarly to class `Server`, except that the `Client` sends packets only when the user types a message in a text field and presses the *Enter* key.

When this occurs, the program calls method `actionPerformed` (Fig. 28.9, lines 32–57), which converts the `String` the user entered into a `byte` array (line 41). Lines 44–45 create a `DatagramPacket` and initialize it with the `byte` array, the length of the `String` that was entered by the user, the IP address to which the packet is to be sent (`InetAddress.getLocalHost()` in this example) and the port number at which the `Server` is waiting for packets (5000 in this example). Line 47 sends the packet. The client in this example must know that the server is receiving packets at port 5000—otherwise, the server will *not* receive the packets.

The `DatagramSocket` constructor call (Fig. 28.9, line 71) in this application does not specify any arguments. This no-argument constructor allows the computer to select the next available port number for the `DatagramSocket`. The client does not need a specific port number, because the server receives the client's port number as part of each `DatagramPacket` sent by the client. Thus, the server can send packets back to the same computer and port number from which it receives a packet of information.

```
1  // Fig. 28.9: Client.java
2  // Client side of connectionless client/server computing with datagrams.
3  import java.io.IOException;
4  import java.net.DatagramPacket;
5  import java.net.DatagramSocket;
6  import java.net.InetAddress;
7  import java.net.SocketException;
8  import java.awt.BorderLayout;
9  import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import javax.swing.JFrame;
12 import javax.swing.JScrollPane;
13 import javax.swing.JTextArea;
14 import javax.swing.JTextField;
15 import javax.swing.SwingUtilities;
16
17 public class Client extends JFrame
18 {
19    private JTextField enterField; // for entering messages
20    private JTextArea displayArea; // for displaying messages
21    private DatagramSocket socket; // socket to connect to server
22
23    // set up GUI and DatagramSocket
24    public Client()
25    {
26       super("Client");
27
28       enterField = new JTextField("Type message here");
29       enterField.addActionListener(
30          new ActionListener()
31          {
32             public void actionPerformed(ActionEvent event)
33             {
34                try // create and send packet
35                {
```

Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 1 of 3.)

```java
                  // get message from textfield
                  String message = event.getActionCommand();
                  displayArea.append("\nSending packet containing: " +
                     message + "\n");

                  byte[] data = message.getBytes(); // convert to bytes

                  // create sendPacket
                  DatagramPacket sendPacket = new DatagramPacket(data,
                     data.length, InetAddress.getLocalHost(), 5000);

                  socket.send(sendPacket); // send packet
                  displayArea.append("Packet sent\n");
                  displayArea.setCaretPosition(
                     displayArea.getText().length());
               }
               catch (IOException ioException)
               {
                  displayMessage(ioException + "\n");
                  ioException.printStackTrace();
               }
            }
         }
      );

      add(enterField, BorderLayout.NORTH);

      displayArea = new JTextArea();
      add(new JScrollPane(displayArea), BorderLayout.CENTER);

      setSize(400, 300); // set window size
      setVisible(true); // show window

      try // create DatagramSocket for sending and receiving packets
      {
         socket = new DatagramSocket();
      }
      catch (SocketException socketException)
      {
         socketException.printStackTrace();
         System.exit(1);
      }
   }

   // wait for packets to arrive from Server, display packet contents
   public void waitForPackets()
   {
      while (true)
      {
         try // receive packet and display contents
         {
            byte[] data = new byte[100]; // set up packet
```

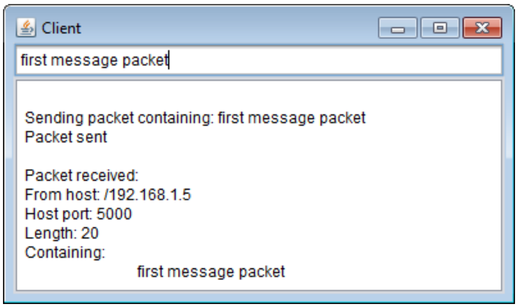
Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 2 of 3.)

```java
            DatagramPacket receivePacket = new DatagramPacket(
               data, data.length);

            socket.receive(receivePacket); // wait for packet

            // display packet contents
            displayMessage("\nPacket received:" +
               "\nFrom host: " + receivePacket.getAddress() +
               "\nHost port: " + receivePacket.getPort() +
               "\nLength: " + receivePacket.getLength() +
               "\nContaining:\n\t" + new String(receivePacket.getData(),
                  0, receivePacket.getLength()));
         }
         catch (IOException exception)
         {
            displayMessage(exception + "\n");
            exception.printStackTrace();
         }
      }
   }

   // manipulates displayArea in the event-dispatch thread
   private void displayMessage(final String messageToDisplay)
   {
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates displayArea
            {
               displayArea.append(messageToDisplay);
            }
         }
      );
   }
}
```

Fig. 28.9 | Client side of connectionless client/server computing with datagrams. (Part 3 of 3.)

```java
// Fig. 28.10: ClientTest.java
// Tests the Client class.
import javax.swing.JFrame;

public class ClientTest
{
   public static void main(String[] args)
   {
      Client application = new Client(); // create client
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.waitForPackets(); // run client application
   }
}
```

Fig. 28.10 | Class that tests the `Client`. (Part 1 of 2.)



`Client` window after sending packet to `Server` and receiving packet back from `Server`

Fig. 28.10 | Class that tests the `Client`. (Part 2 of 2.)

Method `waitForPackets`

`Client` method `waitForPackets` (lines 81–107) uses an infinite loop to wait for packets from the server. Line 91 blocks until a packet arrives. This does not prevent the user from sending a packet, because the *GUI events are handled in the event-dispatch thread*. It only prevents the `while` loop from continuing until a packet arrives at the `Client`. When a packet arrives, line 91 stores it in `receivePacket`, and lines 94–99 call method `display-Message` (declared at lines 110–121) to display the packet's contents in the text area.

28.7 Client/Server Tic-Tac-Toe Using a Multithreaded Server

This section presents the popular game Tic-Tac-Toe implemented by using client/server techniques with stream sockets. The program consists of a `TicTacToeServer` application (Figs. 28.11–28.12) that allows two `TicTacToeClient` applications (Figs. 28.13–28.14) to connect to the server and play Tic-Tac-Toe. Sample outputs are shown in Fig. 28.15.

`TicTacToeServer` *Class*

As the `TicTacToeServer` receives each client connection, it creates an instance of inner-class `Player` (Fig. 28.11, lines 182–304) to process the client in a *separate thread*. These threads enable the clients to play the game independently. The first client to connect to the server is player X and the second is player O. Player X makes the first move. The server maintains the information about the board so it can determine if a player's move is valid.

```
1  // Fig. 28.11: TicTacToeServer.java
2  // Server side of client/server Tic-Tac-Toe program.
3  import java.awt.BorderLayout;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.io.IOException;
7  import java.util.Formatter;
8  import java.util.Scanner;
9  import java.util.concurrent.ExecutorService;
10 import java.util.concurrent.Executors;
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 1 of 7.)

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
import javax.swing.JFrame;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;

public class TicTacToeServer extends JFrame
{
   private String[] board = new String[9]; // tic-tac-toe board
   private JTextArea outputArea; // for outputting moves
   private Player[] players; // array of Players
   private ServerSocket server; // server socket to connect with clients
   private int currentPlayer; // keeps track of player with current move
   private final static int PLAYER_X = 0; // constant for first player
   private final static int PLAYER_O = 1; // constant for second player
   private final static String[] MARKS = { "X", "O" }; // array of marks
   private ExecutorService runGame; // will run players
   private Lock gameLock; // to lock game for synchronization
   private Condition otherPlayerConnected; // to wait for other player
   private Condition otherPlayerTurn; // to wait for other player's turn

   // set up tic-tac-toe server and GUI that displays messages
   public TicTacToeServer()
   {
      super("Tic-Tac-Toe Server"); // set title of window

      // create ExecutorService with a thread for each player
      runGame = Executors.newFixedThreadPool(2);
      gameLock = new ReentrantLock(); // create lock for game

      // condition variable for both players being connected
      otherPlayerConnected = gameLock.newCondition();

      // condition variable for the other player's turn
      otherPlayerTurn = gameLock.newCondition();

      for (int i = 0; i < 9; i++)
         board[i] = new String(""); // create tic-tac-toe board
      players = new Player[2]; // create array of players
      currentPlayer = PLAYER_X; // set current player to first player

      try
      {
         server = new ServerSocket(12345, 2); // set up ServerSocket
      }
      catch (IOException ioException)
      {
         ioException.printStackTrace();
         System.exit(1);
      }

      outputArea = new JTextArea(); // create JTextArea for output
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 2 of 7.)

```java
      add(outputArea, BorderLayout.CENTER);
      outputArea.setText("Server awaiting connections\n");

      setSize(300, 300); // set size of window
      setVisible(true); // show window
   }

   // wait for two connections so game can be played
   public void execute()
   {
      // wait for each client to connect
      for (int i = 0; i < players.length; i++)
      {
         try // wait for connection, create Player, start runnable
         {
            players[i] = new Player(server.accept(), i);
            runGame.execute(players[i]); // execute player runnable
         }
         catch (IOException ioException)
         {
            ioException.printStackTrace();
            System.exit(1);
         }
      }

      gameLock.lock(); // lock game to signal player X's thread

      try
      {
         players[PLAYER_X].setSuspended(false); // resume player X
         otherPlayerConnected.signal(); // wake up player X's thread
      }
      finally
      {
         gameLock.unlock(); // unlock game after signalling player X
      }
   }

   // display message in outputArea
   private void displayMessage(final String messageToDisplay)
   {
      // display message from event-dispatch thread of execution
      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run() // updates outputArea
            {
               outputArea.append(messageToDisplay); // add message
            }
         }
      );
   }
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 3 of 7.)

```java

   // determine if move is valid
   public boolean validateAndMove(int location, int player)
   {
      // while not current player, must wait for turn
      while (player != currentPlayer)
      {
         gameLock.lock(); // lock game to wait for other player to go

         try
         {
            otherPlayerTurn.await(); // wait for player's turn
         }
         catch (InterruptedException exception)
         {
            exception.printStackTrace();
         }
         finally
         {
            gameLock.unlock(); // unlock game after waiting
         }
      }

      // if location not occupied, make move
      if (!isOccupied(location))
      {
         board[location] = MARKS[currentPlayer]; // set move on board
         currentPlayer = (currentPlayer + 1) % 2; // change player

         // let new current player know that move occurred
         players[currentPlayer].otherPlayerMoved(location);

         gameLock.lock(); // lock game to signal other player to go

         try
         {
            otherPlayerTurn.signal(); // signal other player to continue
         }
         finally
         {
            gameLock.unlock(); // unlock game after signaling
         }

         return true; // notify player that move was valid
      }
      else // move was not valid
         return false; // notify player that move was invalid
   }

   // determine whether location is occupied
   public boolean isOccupied(int location)
   {
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 4 of 7.)

```java
      if (board[location].equals(MARKS[PLAYER_X]) ||
         board [location].equals(MARKS[PLAYER_O]))
         return true; // location is occupied
      else
         return false; // location is not occupied
   }

   // place code in this method to determine whether game over
   public boolean isGameOver()
   {
      return false; // this is left as an exercise
   }

   // private inner class Player manages each Player as a runnable
   private class Player implements Runnable
   {
      private Socket connection; // connection to client
      private Scanner input; // input from client
      private Formatter output; // output to client
      private int playerNumber; // tracks which player this is
      private String mark; // mark for this player
      private boolean suspended = true; // whether thread is suspended

      // set up Player thread
      public Player(Socket socket, int number)
      {
         playerNumber = number; // store this player's number
         mark = MARKS[playerNumber]; // specify player's mark
         connection = socket; // store socket for client

         try // obtain streams from Socket
         {
            input = new Scanner(connection.getInputStream());
            output = new Formatter(connection.getOutputStream());
         }
         catch (IOException ioException)
         {
            ioException.printStackTrace();
            System.exit(1);
         }
      }

      // send message that other player moved
      public void otherPlayerMoved(int location)
      {
         output.format("Opponent moved\n");
         output.format("%d\n", location); // send location of move
         output.flush(); // flush output
      }

```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 5 of 7.)

```
      // control thread's execution
      public void run()
      {
         // send client its mark (X or O), process messages from client
         try
         {
            displayMessage("Player " + mark + " connected\n");
            output.format("%s\n", mark); // send player's mark
            output.flush(); // flush output

            // if player X, wait for another player to arrive
            if (playerNumber == PLAYER_X)
            {
               output.format("%s\n%s", "Player X connected",
                  "Waiting for another player\n");
               output.flush(); // flush output

               gameLock.lock(); // lock game to  wait for second player

               try
               {
                  while(suspended)
                  {
                     otherPlayerConnected.await(); // wait for player O
                  }
               }
               catch (InterruptedException exception)
               {
                  exception.printStackTrace();
               }
               finally
               {
                  gameLock.unlock(); // unlock game after second player
               }

               // send message that other player connected
               output.format("Other player connected. Your move.\n");
               output.flush(); // flush output
            }
            else
            {
               output.format("Player O connected, please wait\n");
               output.flush(); // flush output
            }

            // while game not over
            while (!isGameOver())
            {
               int location = 0; // initialize move location

               if (input.hasNext())
                  location = input.nextInt(); // get move location

```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 6 of 7.)

```java
               // check for valid move
               if (validateAndMove(location, playerNumber))
               {
                  displayMessage("\nlocation: " + location);
                  output.format("Valid move.\n"); // notify client
                  output.flush(); // flush output
               }
               else // move was invalid
               {
                  output.format("Invalid move, try again\n");
                  output.flush(); // flush output
               }
            }
         }
         finally
         {
            try
            {
               connection.close(); // close connection to client
            }
            catch (IOException ioException)
            {
               ioException.printStackTrace();
               System.exit(1);
            }
         }
      }

      // set whether or not thread is suspended
      public void setSuspended(boolean status)
      {
         suspended = status; // set value of suspended
      }
   }
}
```

Fig. 28.11 | Server side of client/server Tic-Tac-Toe program. (Part 7 of 7.)

```java
// Fig. 28.12: TicTacToeServerTest.java
// Class that tests Tic-Tac-Toe server.
import javax.swing.JFrame;

public class TicTacToeServerTest
{
   public static void main(String[] args)
   {
      TicTacToeServer application = new TicTacToeServer();
      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      application.execute();
   }
}
```

Fig. 28.12 | Class that tests Tic-Tac-Toe server. (Part 1 of 2.)

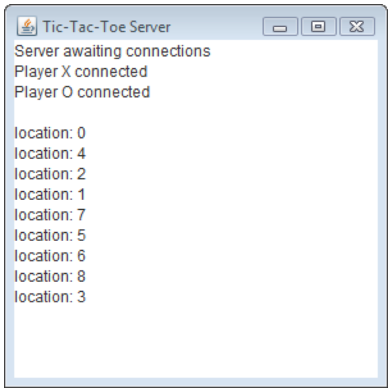


Fig. 28.12 | Class that tests Tic-Tac-Toe server. (Part 2 of 2.)

We begin with a discussion of the server side of the Tic-Tac-Toe game. When the `TicTacToeServer` application executes, the `main` method (lines 7–12 of Fig. 28.12) creates a `TicTacToeServer` object called `application`. The constructor (Fig. 28.11, lines 34–69) attempts to set up a `ServerSocket`. If successful, the program displays the server window, then `main` invokes the `TicTacToeServer` method `execute` (lines 72–100). Method `execute` loops twice, blocking at line 79 each time while waiting for a client connection. When a client connects, line 79 creates a new `Player` object to manage the connection as a separate thread, and line 80 executes the `Player` in the `runGame` thread pool.

When the `TicTacToeServer` creates a `Player`, the `Player` constructor (lines 192–208) receives the `Socket` object representing the connection to the client and gets the associated input and output streams. Line 201 creates a `Formatter` (see Chapter 15) by wrapping it around the output stream of the socket. The `Player`'s `run` method (lines 219–297) controls the information that's sent to and received from the client. First, it passes to the client the character that the client will place on the board when a move is made (line 225). Line 226 calls `Formatter` method **`flush`** to force this output to the client. Line 241 suspends player X's thread as it starts executing, because player X can move only after player O connects.

When player O connects, the game can be played, and the `run` method begins executing its `while` statement (lines 264–283). Each iteration of this loop reads an integer (line 269) representing the location where the client wants to place a mark (blocking to wait for input, if necessary), and line 272 invokes the `TicTacToeServer` method `validateAndMove` (declared at lines 118–163) to check the move. If the move is valid, line 275 sends a message to the client to this effect. If not, line 280 sends a message indicating that the move was invalid. The program maintains board locations as numbers from 0 to 8 (0 through 2 for the first row, 3 through 5 for the second row and 6 through 8 for the third row).

Method `validateAndMove` (lines 118–163 in class `TicTacToeServer`) allows only one player at a time to move, thereby preventing them from modifying the state information of the game simultaneously. If the `Player` attempting to validate a move is *not* the current player (i.e., the one allowed to make a move), it's placed in a *wait* state until its turn to move. If the position for the move being validated is already occupied on the board, `validMove` returns `false`. Otherwise, the server places a mark for the player in its local representation of the board (line 142), notifies the other `Player` object (line 146) that a move has been made (so that the client can be sent a message), invokes method `signal` (line 152) so that the waiting `Player` (if there is one) can validate a move and returns `true` (line 159) to indicate that the move is valid.

`TicTacToeClient` Class

Each `TicTacToeClient` application (Figs. 28.13–28.14; sample outputs in Fig. 28.15) maintains its own GUI version of the Tic-Tac-Toe board on which it displays the state of the game. The clients can place a mark only in an empty square. Inner class `Square` (Fig. 28.13, lines 205–261) implements each of the nine squares on the board. When a `TicTacToeClient` begins execution, it creates a `JTextArea` in which messages from the server and a representation of the board using nine `Square` objects are displayed. The `startClient` method (lines 80–100) opens a connection to the server and gets the associated input and output streams from the `Socket` object. Lines 85–86 make a connection to the server. Class `TicTacToeClient` implements interface `Runnable` so that a separate thread can read messages from the server. This approach enables the user to interact with the board (in the event-dispatch thread) while waiting for messages from the server. After establishing the connection to the server, line 99 executes the client with the `worker` `ExecutorService`. The `run` method (lines 103–126) controls the separate thread of execution. The method first reads the mark character (X or O) from the server (line 105), then loops continuously (lines 121–125) and reads messages from the server (line 124). Each message is passed to the `processMessage` method (lines 129–156) for processing.

```
1   // Fig. 28.13: TicTacToeClient.java
2   // Client side of client/server Tic-Tac-Toe program.
3   import java.awt.BorderLayout;
4   import java.awt.Dimension;
5   import java.awt.Graphics;
6   import java.awt.GridLayout;
7   import java.awt.event.MouseAdapter;
8   import java.awt.event.MouseEvent;
9   import java.net.Socket;
10  import java.net.InetAddress;
11  import java.io.IOException;
12  import javax.swing.JFrame;
13  import javax.swing.JPanel;
14  import javax.swing.JScrollPane;
15  import javax.swing.JTextArea;
16  import javax.swing.JTextField;
17  import javax.swing.SwingUtilities;
18  import java.util.Formatter;
19  import java.util.Scanner;
20  import java.util.concurrent.Executors;
21  import java.util.concurrent.ExecutorService;
22
23  public class TicTacToeClient extends JFrame implements Runnable
24  {
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 1 of 6.)

```java
   private JTextField idField; // textfield to display player's mark
   private JTextArea displayArea; // JTextArea to display output
   private JPanel boardPanel; // panel for tic-tac-toe board
   private JPanel panel2; // panel to hold board
   private Square[][] board; // tic-tac-toe board
   private Square currentSquare; // current square
   private Socket connection; // connection to server
   private Scanner input; // input from server
   private Formatter output; // output to server
   private String ticTacToeHost; // host name for server
   private String myMark; // this client's mark
   private boolean myTurn; // determines which client's turn it is
   private final String X_MARK = "X"; // mark for first client
   private final String O_MARK = "O"; // mark for second client

   // set up user-interface and board
   public TicTacToeClient(String host)
   {
      ticTacToeHost = host; // set name of server
      displayArea = new JTextArea(4, 30); // set up JTextArea
      displayArea.setEditable(false);
      add(new JScrollPane(displayArea), BorderLayout.SOUTH);

      boardPanel = new JPanel(); // set up panel for squares in board
      boardPanel.setLayout(new GridLayout(3, 3, 0, 0));

      board = new Square[3][3]; // create board

      // loop over the rows in the board
      for (int row = 0; row < board.length; row++)
      {
         // loop over the columns in the board
         for (int column = 0; column < board[row].length; column++)
         {
            // create square
            board[row][column] = new Square(' ', row * 3 + column);
            boardPanel.add(board[row][column]); // add square
         }
      }

      idField = new JTextField(); // set up textfield
      idField.setEditable(false);
      add(idField, BorderLayout.NORTH);

      panel2 = new JPanel(); // set up panel to contain boardPanel
      panel2.add(boardPanel, BorderLayout.CENTER); // add board panel
      add(panel2, BorderLayout.CENTER); // add container panel

      setSize(300, 225); // set size of window
      setVisible(true); // show window

      startClient();
   }
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 2 of 6.)

```java

   // start the client thread
   public void startClient()
   {
      try // connect to server and get streams
      {
         // make connection to server
         connection = new Socket(
            InetAddress.getByName(ticTacToeHost), 12345);

         // get streams for input and output
         input = new Scanner(connection.getInputStream());
         output = new Formatter(connection.getOutputStream());
      }
      catch (IOException ioException)
      {
         ioException.printStackTrace();
      }

      // create and start worker thread for this client
      ExecutorService worker = Executors.newFixedThreadPool(1);
      worker.execute(this); // execute client
   }

   // control thread that allows continuous update of displayArea
   public void run()
   {
      myMark = input.nextLine(); // get player's mark (X or O)

      SwingUtilities.invokeLater(
         new Runnable()
         {
            public void run()
            {
               // display player's mark
               idField.setText("You are player \"" + myMark + "\"");
            }
         }
      );

      myTurn = (myMark.equals(X_MARK)); // determine if client's turn

      // receive messages sent to client and output them
      while (true)
      {
         if (input.hasNextLine())
            processMessage(input.nextLine());
      }
   }

   // process messages received by client
   private void processMessage(String message)
   {
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 3 of 6.)

```java
         // valid move occurred
         if (message.equals("Valid move."))
         {
            displayMessage("Valid move, please wait.\n");
            setMark(currentSquare, myMark); // set mark in square
         }
         else if (message.equals("Invalid move, try again"))
         {
            displayMessage(message + "\n"); // display invalid move
            myTurn = true; // still this client's turn
         }
         else if (message.equals("Opponent moved"))
         {
            int location = input.nextInt(); // get move location
            input.nextLine(); // skip newline after int location
            int row = location / 3; // calculate row
            int column = location % 3; // calculate column

            setMark( board[row][column],
               (myMark.equals(X_MARK) ? O_MARK : X_MARK)); // mark move
            displayMessage("Opponent moved. Your turn.\n");
            myTurn = true; // now this client's turn
         }
         else
            displayMessage(message + "\n"); // display the message
      }

      // manipulate displayArea in event-dispatch thread
      private void displayMessage(final String messageToDisplay)
      {
         SwingUtilities.invokeLater(
            new Runnable()
            {
               public void run()
               {
                  displayArea.append(messageToDisplay); // updates output
               }
            }
         );
      }

      // utility method to set mark on board in event-dispatch thread
      private void setMark(final Square squareToMark, final String mark)
      {
         SwingUtilities.invokeLater(
            new Runnable()
            {
               public void run()
               {
                  squareToMark.setMark(mark); // set mark in square
               }
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 4 of 6.)

```java
         }
      );
   }

   // send message to server indicating clicked square
   public void sendClickedSquare(int location)
   {
      // if it is my turn
      if (myTurn)
      {
         output.format("%d\n", location); // send location to server
         output.flush();
         myTurn = false; // not my turn any more
      }
   }

   // set current Square
   public void setCurrentSquare(Square square)
   {
      currentSquare = square; // set current square to argument
   }

   // private inner class for the squares on the board
   private class Square extends JPanel
   {
      private String mark; // mark to be drawn in this square
      private int location; // location of square

      public Square(String squareMark, int squareLocation)
      {
         mark = squareMark; // set mark for this square
         location = squareLocation; // set location of this square

         addMouseListener(
            new MouseAdapter()
            {
               public void mouseReleased(MouseEvent e)
               {
                  setCurrentSquare(Square.this); // set current square

                  // send location of this square
                  sendClickedSquare(getSquareLocation());
               }
            }
         );
      }

      // return preferred size of Square
      public Dimension getPreferredSize()
      {
         return new Dimension(30, 30); // return preferred size
      }
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 5 of 6.)

```java

      // return minimum size of Square
      public Dimension getMinimumSize()
      {
         return getPreferredSize(); // return preferred size
      }

      // set mark for Square
      public void setMark(String newMark)
      {
         mark = newMark; // set mark of square
         repaint(); // repaint square
      }

      // return Square location
      public int getSquareLocation()
      {
         return location; // return location of square
      }

      // draw Square
      public void paintComponent(Graphics g)
      {
         super.paintComponent(g);

         g.drawRect(0, 0, 29, 29); // draw square
         g.drawString(mark, 11, 20); // draw mark
      }
   }
}
```

Fig. 28.13 | Client side of client/server Tic-Tac-Toe program. (Part 6 of 6.)

```java
// Fig. 28.14: TicTacToeClientTest.java
// Test class for Tic-Tac-Toe client.
import javax.swing.JFrame;

public class TicTacToeClientTest
{
   public static void main(String[] args)
   {
      TicTacToeClient application; // declare client application

      // if no command line args
      if (args.length == 0)
         application = new TicTacToeClient("127.0.0.1"); // localhost
      else
         application = new TicTacToeClient(args[0]); // use args

      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }
}
```

Fig. 28.14 | Test class for Tic-Tac-Toe client.

If the message received is `"Valid move."`, lines 134–135 display the message `"Valid move, please wait."` and call method `setMark` (lines 173–184) to set the client's mark in the current square (the one in which the user clicked), using `SwingUtilities` method `invokeLater` to ensure that the GUI updates occur in the event-dispatch thread. If the message received is `"Invalid move, try again."`, line 139 displays the message so that the user can click a different square. If the message received is `"Opponent moved."`, line 144 reads an integer from the server indicating where the opponent moved, and lines 149–150 place a mark in that square of the board (again using `SwingUtilities` method `invokeLater` to ensure that the GUI updates occur in the event-dispatch thread). If any other message is received, line 155 simply displays the message.

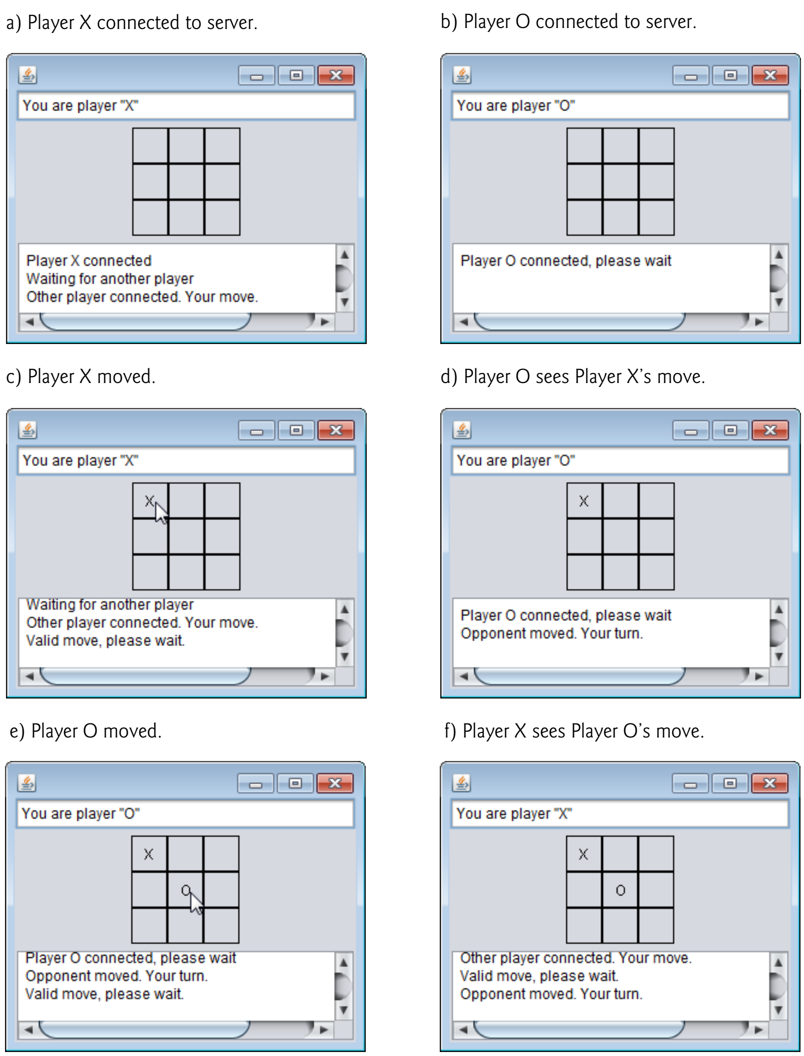


Fig. 28.15 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 1 of 2.)

g) Player X moved. h) Player O sees Player X's last move.

Fig. 28.15 | Sample outputs from the client/server Tic-Tac-Toe program. (Part 2 of 2.)

28.8 Optional Online Case Study: `DeitelMessenger`[2]

This case study is available at `http://www.deitel.com/books/jhtp11`. Chat rooms provide a central location where users can chat with each other via short text messages. Each participant can see all the messages that the other users post, and each user can post messages. This case study integrates many of the Java networking, multithreading and Swing GUI features you've learned thus far to build an online chat system. We also introduce **multicasting**, which enables an application to send `DatagramPacket`s to *groups* of clients.

The `DeitelMessenger` case study is a significant application that uses many intermediate Java features, such as networking with `Socket`s, `DatagramPacket`s and `MulticastSocket`s, multithreading and Swing GUI. The case study also demonstrates good software engineering practices by separating interface from implementation and enabling developers to support different network protocols and provide different user interfaces. After reading this case study, you'll be able to build more significant networking applications.

28.9 Wrap-Up

In this chapter, you learned the basics of network programming in Java. You learned two different methods of sending data over a network—streams-based networking using TCP/IP and datagrams-based networking using UDP. We showed how to build simple client/server chat programs using both streams-based and datagram-based networking. You then saw a client/server Tic-Tac-Toe game that enables two clients to play by interacting with a multithreaded server that maintains the game's state and logic. In the next chapter, you'll learn basic database concepts, how to interact with data in a database using SQL and how to use JDBC to allow Java applications to manipulate database data.

2. This case study is from the Seventh Edition of this book and is provided as is. We no longer provide support for it.

Summary

Section 28.1 Introduction

- Java provides stream sockets and datagram sockets (p. 2). With stream sockets (p. 2), a process establishes a connection (p. 2) to another process. While the connection is in place, data flows between the processes in streams. Stream sockets are said to provide a connection-oriented service (p. 2). The protocol used for transmission is the popular TCP (Transmission Control Protocol; p. 2).
- With datagram sockets (datagram socket), individual packets of information are transmitted. UDP (User Datagram Protocol; p. 2) is a connectionless service that does not guarantee that packets will not be lost, duplicated or arrive out of sequence.

Section 28.2 Reading a File on a Web Server

- `JEditorPane` (p. 3) method `setPage` (p. 5) downloads the document specified by its argument and displays it.
- Typically, an HTML document contains hyperlinks that link to other documents on the web. If an HTML document is displayed in an uneditable `JEditorPane` and the user clicks a hyperlink (p. 5), a `HyperlinkEvent` (p. 5) occurs and the `HyperlinkListeners` are notified.
- `HyperlinkEvent` method `getEventType` (p. 5) determines the event type. `HyperlinkEvent` contains nested class `EventType` (p. 5), which declares event types `ACTIVATED`, `ENTERED` and `EXITED`. `HyperlinkEvent` method `getURL` (p. 5) obtains the `URL` represented by the hyperlink.

Section 28.3 Establishing a Simple Server Using Stream Sockets

- Stream-based connections (p. 2) are managed with `Socket` objects (p. 6).
- A `ServerSocket` object (p. 6) establishes the port (p. 6) where a server (p. 2) waits for connections from clients (p. 2). `ServerSocket` method `accept` (p. 6) waits indefinitely for a connection from a client and returns a `Socket` object when a connection is established.
- `Socket` methods `getOutputStream` and `getInputStream` (p. 7) get references to a `Socket`'s `OutputStream` and `InputStream`, respectively. Method `close` (p. 7) terminates a connection.

Section 28.4 Establishing a Simple Client Using Stream Sockets

- A server name and port number (p. 6) are specified when creating a `Socket` object to enable it to connect a client to the server. A failed connection attempt throws an `IOException`.
- `InetAddress` method `getByName` (p. 20) returns an `InetAddress` object (p. 14) containing the IP address of the specified computer. `InetAddress` method `getLocalHost` (p. 20) returns an `InetAddress` object containing the IP address of the local computer executing the program.

Section 28.6 Datagrams: Connectionless Client/Server Interaction

- Connection-oriented transmission is like the telephone system—you dial and are given a connection to the telephone of the person with whom you wish to communicate. The connection is maintained for the duration of your phone call, even when you aren't talking.
- Connectionless transmission (p. 20) with datagrams is similar to mail carried via the postal service. A large message that will not fit in one envelope can be broken into separate message pieces that are placed in separate, sequentially numbered envelopes. All the letters are then mailed at once. They could arrive in order, out of order or not at all.
- `DatagramPacket` objects store packets of data that are to be sent or that are received by an application. `DatagramSockets` send and receive `DatagramPackets`.

- The `DatagramSocket` constructor that takes no arguments binds the `DatagramSocket` to a port chosen by the computer executing the program. The one that takes an integer port-number argument binds the `DatagramSocket` to the specified port. If a `DatagramSocket` constructor fails to bind the `DatagramSocket` to a port, a `SocketException` occurs (p. 21). `DatagramSocket` method `receive` (p. 24) blocks (waits) until a packet arrives, then stores the packet in its argument.
- `DatagramPacket` method `getAddress` (p. 24) returns an `InetAddress` object containing information about the computer from or to which the packet was sent. Method `getPort` (p. 24) returns an integer specifying the port number (p. 6) through which the `DatagramPacket` was sent or received. Method `getLength` (`getLength`) returns the number of bytes of data in a `DatagramPacket`. Method `getData` (p. 24) returns a byte array containing the data.
- The `DatagramPacket` constructor for a packet to be sent takes four arguments—the byte array to be sent, the number of bytes to be sent, the client address to which the packet will be sent and the port number where the client is waiting to receive packets.
- `DatagramSocket` method `send` (p. 24) sends a `DatagramPacket` out over the network.
- If an error occurs when receiving or sending a `DatagramPacket`, an `IOException` occurs.

Self-Review Exercises

28.1 Fill in the blanks in each of the following statements:

a) Exception ________ occurs when an input/output error occurs when closing a socket.
b) Exception ________ occurs when a hostname indicated by a client cannot be resolved to an address.
c) If a `DatagramSocket` constructor fails to set up a `DatagramSocket` properly, an exception of type ________ occurs.
d) Many of Java's networking classes are contained in package ________.
e) Class ________ binds the application to a port for datagram transmission.
f) An object of class ________ contains an IP address.
g) The two types of sockets we discussed in this chapter are ________ and ________.
h) Method `getLocalHost` returns a(n) ________ object containing the local IP address of the computer on which the program is executing.
i) The `URL` constructor determines whether its `String` argument is a valid URL. If so, the `URL` object is initialized with that location. If not, a(n) ________ exception occurs.

28.2 State whether each of the following is *true or false*. If *false*, explain why.

a) UDP is a connection-oriented protocol.
b) With stream sockets a process establishes a connection to another process.
c) A server waits at a port for connections from a client.
d) Datagram packet transmission over a network is reliable—packets are guaranteed to arrive in sequence.

Answers to Self-Review Exercises

28.1 a) `IOException`. b) `UnknownHostException`. c) `SocketException`. d) `java.net`. e) `DatagramSocket`. f) `InetAddress`. g) stream sockets, datagram sockets. h) `InetAddress`. i) `MalformedURLException`.

28.2 a) False; UDP is a connectionless protocol and TCP is a connection-oriented protocol. b) True. c) True. d) False; packets can be lost, arrive out of order or be duplicated.

Exercises

28.3 Distinguish between connection-oriented and connectionless network services.

28.4 How does a client determine the hostname of the client computer?

28.5 Under what circumstances would a `SocketException` be thrown?

28.6 How can a client get a line of text from a server?

28.7 Describe how a client connects to a server.

28.8 Describe how a server sends data to a client.

28.9 Describe how to prepare a server to receive a stream-based connection from a single client.

28.10 How does a server listen for streams-based socket connections at a port?

28.11 What determines how many connect requests from clients can wait in a queue to connect to a server?

28.12 As described in the text, what reasons might cause a server to refuse a connection request from a client?

28.13 Use a socket connection to allow a client to specify a filename of a text file and have the server send the contents of the file or indicate that the file does not exist.

28.14 Modify Exercise 28.13 to allow the client to modify the contents of the file and send the file back to the server for storage. The user can edit the file in a `JTextArea`, then click a *save changes* button to send the file back to the server.

28.15 *(Multithreaded Server)* Multithreaded servers are quite popular today, especially because of the increasing use of multi-core servers. Modify the simple server application presented in Section 28.5 to be a multithreaded server. Then use several client applications and have each of them connect to the server simultaneously. Use an `ArrayList` to store the client threads. `ArrayList` provides several methods to use in this exercise. Method `size` determines the number of elements in an `ArrayList`. Method `get` returns the element in the location specified by its argument. Method `add` places its argument at the end of the `ArrayList`. Method `remove` deletes its argument from the `ArrayList`.

28.16 *(Checkers Game)* In the text, we presented a Tic-Tac-Toe program controlled by a multithreaded server. Develop a checkers program modeled after the Tic-Tac-Toe program. The two users should alternate making moves. Your program should mediate the players' moves, determining whose turn it is and allowing only valid moves. The players themselves will determine when the game is over.

28.17 *(Chess Game)* Develop a chess-playing program modeled after Exercise 28.16.

28.18 *(Blackjack Game)* Develop a blackjack card game program in which the server application deals cards to each of the clients. The server should deal additional cards (per the rules of the game) to each player as requested.

28.19 *(Poker Game)* Develop a poker game in which the server application deals cards to each client. The server should deal additional cards (per the rules of the game) to each player as requested.

28.20 *(Modifications to the Multithreaded Tic-Tac-Toe Program)* The programs in Figs. 28.11 and 28.13 implemented a multithreaded, client/server version of the game of Tic-Tac-Toe. Our goal in developing this game was to demonstrate a multithreaded server that could process multiple connections from clients at the same time. The server in the example is really a mediator between the two clients—it makes sure that each move is valid and that each client moves in the proper order. The server does not determine who won or lost or whether there was a draw. Also, there's no capability to allow a new game to be played or to terminate an existing game.

The following is a list of suggested modifications to Figs. 28.11 and 28.13:

a) Modify the `TicTacToeServer` class to test for a win, loss or draw after each move. Send a message to each client that indicates the result of the game when the game is over.
b) Modify the `TicTacToeClient` class to display a button that when clicked allows the client to play another game. The button should be enabled only when a game completes. Both class `TicTacToeClient` and class `TicTacToeServer` must be modified to reset the board and all state information. Also, the other `TicTacToeClient` should be notified that a new game is about to begin so that its board and state can be reset.
c) Modify the `TicTacToeClient` class to provide a button that allows a client to terminate the program at any time. When the user clicks the button, the server and the other client should be notified. The server should then wait for a connection from another client so that a new game can begin.
d) Modify the `TicTacToeClient` class and the `TicTacToeServer` class so that the winner of a game can choose game piece X or O for the next game. Remember: X always goes first.
e) If you'd like to be ambitious, allow a client to play against the server while the server waits for a connection from another client.

28.21 *(3-D Multithreaded Tic-Tac-Toe)* Modify the multithreaded, client/server Tic-Tac-Toe program to implement a three-dimensional 4-by-4-by-4 version of the game. Implement the server application to mediate between the two clients. Display the three-dimensional board as four boards containing four rows and four columns each. If you're ambitious, try the following modifications:
a) Draw the board in a three-dimensional manner.
b) Allow the server to test for a win, loss or draw. Beware! There are many possible ways to win on a 4-by-4-by-4 board!

28.22 *(Networked Morse Code)* Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit, and a few special characters (e.g., period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound and the dash a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems. Separation between words is indicated by a space or, simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short time during which no sound is transmitted. The international version of the Morse code appears in Fig. 28.16.

Character	Code	Character	Code	Character	Code	Character	Code
A	.-	J	.---	S	...	1	.----
B	-...	K	-.-	T	-	2	..---
C	-.-.	L	.-..	U	..-	3	...--
D	-..	M	--	V	...-	4	....-
E	.	N	-.	W	.--	5	.....
F	..-.	O	---	X	-..-	6	-....
G	--.	P	.--.	Y	-.--	7	--...
H	....	Q	--.-	Z	--..	8	---..
I	..	R	.-.			9	----.
						0	-----

Fig. 28.16 | Letters and digits in international Morse code.

Write a client/server application in which two clients can send Morse-code messages to each other through a multithreaded server application. The client application should allow the user to type English-language phrases in a `JTextArea`. When the user sends the message, the client application encodes the text into Morse code and sends the coded message through the server to the other client. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word. When messages are received, they should be decoded and displayed as normal characters and as Morse code. The client should have one `JTextField` for typing and one `JTextArea` for displaying the other client's messages.